A Review on Bad Smell Code and Software Refactoring Process

Shubham P. Chavhan, Rutika S. Chaudhari, Priyanka P. Chavhan
Student, Department of CSE, BNCOE, Pusad, Maharashtra,
India

DOI: 10.23956/ijarcsse/V7I4/0131

Abstract—There is a constant need for practical, efficient, and cost effective software evaluation techniques. As the application code becomes older & older, maintaining it becomes a challenge for the enterprises due to increased cost of any further change in it. Refactoring is a technique to keep the code cleaner, simpler, extendable, reusable and maintainable. Refactoring leads to constant improvement in software quality while providing reusable, modular and service oriented components. It is a disciplined and controlled technique for improving the software code by changing the internal structure of code without affecting the functionalities. However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck (Fowler & Beck 2000) give a list of bad code smells. Bad smells are signs of potential problems in code. To this end, we propose a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution.

Keywords—Code clean up, code standard, maintainability, extendibility, software refactoring, bad smell code.

I. INTRODUCTION

As the application code becomes older & older, maintaining it becomes a challenge for the enterprises due to increased cost of any further change in it. "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure." Refactoring is typically done in small steps. After each small step, we're left with a working system that's functionally unchanged. Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probably the most important. To refactor programming code is to rewrite the code, to "clean it up", refactoring is the moving of units of functionality from one place to another in your program, it has as a primary objective, getting each piece of functionality to exist in exactly one place in the software. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code. Practically, refactoring means making code cleaner, cleaner, simpler and elegant or, in other words, clean up after our self when we code. Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probably the most important. To refactor programming code is to rewrite the code, to “clean it up”. Refactoring is the moving of units of functionality from one place to another in your program. Refactoring has as a primary objective, getting each piece of functionality to exist in exactly one place in the software Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. A key issue in software refactoring is determining the kind of source code that requires refactoring. Experts have summarized typical situations that may require refactoring (Martin Fowler, 1999), (W.C. Wake et al, 2003). Fowler et al.call them Bad Smells (Martin Fowler, 1999), indicating that some part of the source code smells terrible. In other words, Bad Smells (e.g., Duplicated Code) are signs of potential problems in code that may require refactoring. These bad smells are usually linked to corresponding refactoring rules that can help dispel these bad smells.

II. REFACTORING ACTIVITIES

The refactoring process consists of a number of different activities, each of which can be automated to a certain extent:

- Identify where the software should be refactored
- Determine which refactoring should be applied to the identified places;
- Guarantee that the applied refactoring preserves behavior;
- Apply the refactoring; d bugs .It helps in finding the Bugs present in the program. Refactoring helps to program fast.
- Assess the effect of refactoring on software quality characteristics;
III. REFACTORING PROCESS

Fig 1: Process of the software refactoring

IV. REFACTORING TECHNIQUES

4.1 Extract Method:
Extraction of a piece of code into a separate method. You have a code fragment that appears in multiple places within the code. You have a code fragment that can be grouped together. Turn this fragment into a method whose name explains the purpose of this method.

```java
void printOwing() {
    printBanner();
    //print details System.out.println ("name: "+_name);
    System.out.println ("amount "+getOutstanding());
}
```

4.2. Replace Temp with Query:
Replacement of references to the temporary variable with the method calls. It facilitates method extraction. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

```java
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.95
```

```java
double basePrice = _quantity * _itemPrice;
if (basePrice ( ) > 1000)
    return basePrice ( ) * 0.95;
else
    return basePrice ( ) * 0.95
```
return basePrice() * 0.98; ... double basePrice()
{
    return _quantity*_itemPrice;
}

4.3 Inline Method:
A method’s body is just as clear as its name. Put the method’s body into the body of its callers and remove the method.[4]

    int getRating()
    {
        return(moreThanFiveLateDeliveries()) ? 2 : 1;
    }

    boolean moreThanFiveLateDeliveries()
    {
        return _numberOfLateDeliveries;
    }

    int getRating()
    {
        return(_numberOfLateDeliveries > 5) ? 2 : 1;
    }

This refactoring is opposite to Extract Method. It will inline all invocations of a certain method and remove the method declaration. When invoked on a particular method’s usage, there is an option to inline only this usage and keep the method. This refactoring is often useful as a part of complex refactorings.

4.4 Replace Array With Object:
You have an array in which certain elements mean different things. Replace the array with an object that has a field for each element.

    String[] row = new String[3];
    row [0] = "Liverpool" ;
    row [1] = "15";

    Performance row = new Performance();
    row.setName("Liverpool");
    row.setWins("15");

V. TYPES OF SMELLS

5.1 Long Method:
Is a method that is too long, so it is difficult to understand, change, or extend. Fowler and beck (fowler & beck 2000) strongly believe in short methods. The longer a procedure is the more difficult it is to understand. Nearly all of the time all you have to do to shorten a method is extract method.

Long method can be summarized –
- Symptoms: too long method that is difficult to understand and reuse
- Detection: cyclomatic complexity (polynomial metrics)
- Relationship: n/a
- Solutions: decompose conditional, extract method, replace method with method object.
- Identifications: medium

5.2 Large class:
Means that a class is trying to do too much. These classes have too many instance variables or methods, duplicated code cannot be far behind. A class with too much code is also a breeding ground for duplication. In both cases extract class and extract subclass will work.

This smell can be summarized as follows –
- Symptoms: too many instance variables or methods
- Detection: lack of cohesion methods or measuring class size
- Relationship: blob/god object
- Solutions: extract class, extract interface, extract subclass, and introduce foreign method
- Identifications: medium
- Removal: difficult
- Impact: strong
5.3 Primitive obsession:
Smell represents a case where primitives are used instead of small classes. For example, to represent money, programmers use primitives rather than creating a separate class that could encapsulate the necessary functionality like currency conversion.

Primitive obsession has the following features:
- **Symptoms:** using primitive instead of small classes
- **Detection:** n/a
- **Relationship:** n/a
- **Solutions:** extract class, introduce parameter object, replace array with object, replace data
- **Value with object, replace type code with subclass/state/strategy
- **Identifications:** medium
- **Removal:** medium
- **Impact:** strong

5.4 Long parameter list:
It is a parameter list that is too long and thus difficult to understand. With objects you don’t need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs. This is goodness, because long parameter lists are hard to understand, because they inconsistent and difficult to use because you are forever changing them as you need more data. Use replace parameter with method when you can get the data in one parameter by making a request of an object you already know about.

This smell can be summarized as:
- **Symptoms:** a method with too many parameters that is difficult to understand
- **Detection:** count the number of parameters
- **Relationship:** n/a
- **Solutions:** introduce parameter object, replace method with method object, and preserve whole object
- **Identifications:** medium
- **Removal:** medium

5.5 Data clumps:
Smell means that software has data items that often appear together. Often you will see the same three or four data items together in lots of places:
A) fields in a couple of classes
B) parameters in many method signatures:

Removing one of the group’s data items means that the those items that are left make no sense, e.g., integers specifying RGB colors.

Data clumps has the following features:
- **Symptoms:** data is always coherent with each other.
- **Detection:** if one value is removed, the data set will be meaningless.
- **Relationship:** magic numbers/magic string
- **Solutions:** extract class, introduce parameter object, and preserve whole object
- **Identifications:** medium
- **Removal:** difficult
- **Impact:** medium

5.6 Switch statements:
Smell has a slightly misleading name, since a switch operator does not necessarily imply a smell. The smell means a case where type codes or runtime class type detection are used instead of polymorphism. Also type codes passed on methods are an instance of this smell. “most times you see a switch statement you should consider polymorphism”. Use extract method to extract the switch statement and then move method to get it into the class where the polymorphism is needed. If you only have a few cases that affect a single method then polymorphism is overkill. In this case replace parameter with explicit methods is a good option. If one of your conditional cases is null, try introduce null object.

Switch statements can be summarized as shown below:
- **Symptoms:** replacing polymorphism with type codes or runtime class type detection
- **Detection:** runtime detection
- **Relationship:** n/a
- **Solutions:** introduce null object, replace conditional with polymorphism, replace method with
- **Explicit method, replace type code with subclass/state/strategy
- **Identifications:** medium
- **Removal:** medium
- **Impact:** weak
5.7 Temporary Field:
Smell means that class has a variable which is only used in some situations. Sometimes you will see an object in which an instance variable is set only in certain circumstances. This can make the code difficult to understand because we usually expect an object to use all of its variables. Use Extract Class to create a home for these orphan variables by putting all the code that uses the variable into the component. You can also eliminate conditional code by using Introduce Null Object to create an alternative component for when the variables are not valid. Temporary Field can be summarized as shown below –

- **Symptoms**: A class has a variable that is only used in some situations.
- **Detection**: Comparing different methods that access each field
- **Solutions**: Extract Class and Introduce Null Object.
- **Identifications**: Medium
- **Removal**: Medium
- **Impact**: Weak

5.8 Refused Bequest:
Smell means that a child class does not fully support all the methods or data it inherits. A bad case of this smell exists when the class is refusing to implement an interface. Keeping the Software Maintainable.

Refused Bequest can be summarized as shown below –

- **Symptoms**: A class could not support its inherited methods or inherited data
- **Detection**: N/A
- **Relationship**: N/A
- **Solutions**: Replace Inheritance with Delegation
- **Identifications**: Medium
- **Removal**: Medium

5.9 Alternative Classes with Different Interfaces:
Smell means a case where a class can operate with two alternative classes, but the interface to these alternative classes is different. For example, a class can operate with a ball or a rectangle class, and if it operates with the ball, it calls the method of the ball class playBall() and with the rectangle it calls playRectangle().

5.10 Parallel Inheritance Hierarchies:
Smell means a situation, where two parallel class hierarchies exist and both of these hierarchies must be extended. It is really a special case of shotgun surgery. “Every time you make a subclass of one class, you also have to make a subclass of another”. The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instance of another. If you use Move Method and Move Field, the hierarchy on the referring class disappears.

Parallel Inheritance Hierarchies can be summarized as shown below

- **Symptoms**: Existing parallel class hierarchies
- **Detection**: N/A
- **Relationship**: N/A
- **Solutions**: Move Field and Move Method
- **Identifications**: Medium
- **Removal**: Medium
- **Impact**: Strong

5.11 Lazy class:
Each class you create costs money and time to maintain and understand. Lazy class is a class that is not doing enough and should therefore be removed. “A class that isn’t doing enough to pay for itself should be eliminated”. If you have subclasses that are not doing enough try to use Collapse Hierarchy and nearly useless components should be subjected to Inline Class.

This smell can be summarized as shown below –

- **Symptoms**: A class having little functions
- **Detection**: Measuring the number of fields and methods in conjunction with cyclomatic complexity.
- **Relationship**: Poltergeist/Lava flow
- **Solutions**: Collapse Hierarchy and Inline class
- **Identifications**: Medium
- **Removal**: Easy
- **Impact**: Strong

5.12 Data Class:
Is a class that contains data, but hardly any logic for it. This is bad since classes should contain both data and logic. “These are classes that have fields, getting and setting methods for fields, and nothing else”. These are classes that have
fields, getting and setting methods, and nothing else. Such methods are dumb data holders and are manipulated in far too much detail by other classes. If in a previous life the classes were public fields, apply Encapsulate Field. If you have collection fields, check to see if they are properly encapsulated and apply Encapsulate Collection if they are not. Use Remove Setting Method on any field that should not be changed. Look for where these getters and setters are used by other classes and try to use Move Method to move behavior into the data class. If you can’t move a whole method, use Extract Method to create a method that can be moved.

5.13 Duplicate code:
According to Fowler and Beck (Fowler & Beck 2000), redundant code is the worst smell. We should remove duplicate code whenever we see it, because it means we have to do everything more than once. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them. The simplest duplicated code problem is when you have the same expression in two methods of the same class.

Duplicate code can be summarized as follows –
- Symptoms: Redundant code
- Detection: Percentage of duplicate code lines in the systems
- Relationship: Cut and Paste
- Solutions: Extract Class, Extract Method, Form Template Method, and Pull Up Method
- Identifications: Easy
- Removal: Medium
- Impact: Medium

5.14 Message Chains smell:
This occurs when you see a client that asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, etc. This smell may appear as a long line of get. This methods, or as a sequence of temps. Navigating this way means the client is structured to the structure of the navigation. The move to use in this case is Hide Delegate at various points in the chain. Message chains smell means a case, where a class asks an object from another object, which then asks another and so on. The problem here is that the first class will be coupled to the whole class structure. To reduce this coupling, a middle man can be used. This smell is summarized as follows:
- Symptoms: classes asking object from one to another
- Detection: Measuring the couplings of a method
- Relationship: N/A
- Solutions: Hide Delegate
- Identifications: Medium
- Removal: Medium
- Impact: Strong

5.15 Middle Man:
One the major features of Objects is encapsulation. Encapsulation often comes with delegation. Middle Man smell means that a class is delegating most of its tasks to subsequent classes. Although this is a common pattern in programming, it can hinder the program, if there is too much delegation. The problem here is that every time you need to create new methods or to modify the old ones, you also have to add or modify the delegating method. If only a few methods are not doing much, use Inline Method to inline them into the caller. If there is additional behavior, you can use Replace Delegation with Inheritance to turn the middle man into a subclass of the real object.

Middle Man is summarized as –
- Symptoms: A class delegating most of its tasks to subsequent classes
- Detection: Many methods coupled to one class with a low cyclomatic complexity
- Solutions: Inline Methods, Replace Delegation with Inheritance, and Remove Middleman
- Identifications: Medium
- Removal: Difficult
- Impact: Strong

5.16 Divergent Change:
Smell means that one class needs to be continuously changed for different reasons, e.g., we have to modify the same class whenever we change a database, or add a new calculation formula. Another example is, if you have to change 4 different methods every time the database changes you might have a situation where two objects are better than one. To clean this up you identify everything that changes for a particular cause and use Extract Class to put them all together. Comments are not necessarily a bad smell, but they can be misused to compensate poorly structured code. Often used as deodorant for other smells.

5.17 Dead Code:
The smell has the following features –
- Symptoms: code never process at running time
VI. DETECTION OF BAD SMELL

Software refactoring can be performed in two different ways. The first is xp-style small-step refactoring confined to a few local files. The other is a batch model in which a large system is thoroughly refactored in one attempt in a batch model. Different kinds of bad smells are typically detected and resolved individually. Suppose a software engineer, familiar with a list of bad smells and associated refactoring rules, refactors a large system. He is equipped with bad smell detection tools and automatic or semiautomatic refactoring tools for cleaning up bad smells. He first chooses a detection tool to identify a specific type of bad smell (a detection tool usually uncovers only one kind of bad smell, e.g., clone detection tools are insensitive to bad smells other than clones). The detection tool proposes initial results that require manual confirmation. Once the detected bad smell is confirmed, the software engineer decides how to refactor it. Selected refactoring rules are manually or semi automatically applied to the bad smells with the help of refactoring tools. Then, the software engineer moves on to the next kind of bad smells, and repeats the process until all kinds of bad smells have been detected and resolved. As a result, different kinds of bad smells are detected and resolved one after the other (fig. 2), regardless of whether the sequence is arranged consciously or unconsciously the scheme for bad smells is two-tiered (fig. 2): kind-level and instance-level. The kind-level scheme arranges the detection and resolution sequences of different kinds of bad smells. For example, should large class smells be detected and resolved before the long method type? Instance-level scheme arranges resolution sequences for instances of a specific kind of bad smell (e.g., large class). For example, numerous clones may be detected in a large application, and an instance-level scheme should manage the sequence in which these clones are resolved. In this paper, we focus on the kind-level scheme resolution of one kind of bad smells may influence (simplify or complicate) the detection and resolution of other bad smells. For example, duplicated code may cause long method. Consequently, if duplicated code is resolved first, long method caused by it may disappear as well. However, little is known about the impact of resolving one kind of bad smell on the detection and
resolution of other (remaining) bad smells different resolution sequences of the same set of bad smells may require different efforts because resolving one kind of bad smell may simplify or complicate the detection and resolution of others. Therefore, it is possible to simplify bad smell detection and resolution by arranging appropriate detection and resolution sequences. The sequence in which duplicated code and long method smells are resolved is an excellent example. Resolving duplicated code before long method is easier than the reverse because long method may disappear as a result of resolving duplicated code. Likewise, it is possible to maximize the effect of software refactoring by resolving bad smells in an optimal sequence.

VII. CONCLUSION

Refactoring is a technique to keep the code cleaner, simpler, extendable, reusable and maintainable. Refactoring leads to constant improvement in software quality while providing reusable, modular and service oriented components. Code smell is any symptom that indicating something wrong. It generally indicates that the code should be refactored or the overall design should be re-examined.

REFERENCES

[16] http://books.google.co.in/books?id=1MsETFDPD3l0C&dq=bad+smell+coding+in+software+refactoring+by+martin+fowler+%26+kent+beck&printsec=frontcover&source=ini&hl=en&ei=nrVHTOJDImvQOcx3LAg&sa=X&oi=book_result&ct=result
[18] HELSINKI UNIVERSITY OF TECHNOLOGY, Department of Computer Science and Engineering, Software Business and Engineering Institute-Mika Mäntylä - Bad Smells in Software – a Taxonomy and an Empirical Study